

Negotiation User Guide

K. Hindriks, W.Pasman, D. Tykhonov
27 march 2008

Contents

Negotiation User Guide 1

Contents 2

1. Theory Crash Course 3

Actions 3

Negotiation Protocol 3

Time available for a Session 3

Negotiation Objects 4

Utility of a Bid 5

2. Installing the Environment 6

System Requirements 6

Installation 6

Progress & Error Messages 6

Bug reporting 6

4. Running Negotiations 7

Using an Automatic Agent 8

Using the UIAgent 8

Negotiation Results 9

5. Writing a Negotiation Agent 10

Init 10

ReceiveMessage 11

ChooseAction 11

Compiling an Agent 13

6. Data structures 14

Domain 14

ValueDiscrete (implements Value) 14

IssueDiscrete (implements Issue) 14

UtilitySpace 15

Bid 15

EvaluatorDiscrete (implements Evaluator) 16

References 16

1. Theory Crash Course

This section gives a crash course on some essential theory needed to understand the negotiator system. The first section discusses the negotiation objects that are used in a negotiation. The second section discusses the utility values.

Actions

In negotiation, the two parties take turns in doing the next negotiation action. The possible actions are:

Accept	This action indicates that agent (which usually has the value 'this') accepts the opponent's last bid
Offer	This action indicates that the agent proposes a new bid
EndNegotiation	This action indicates that the agent terminates the entire negotiation, resulting in the lowest possible score of zero for both agents

Negotiation Protocol

The negotiation protocol determines the overall order of actions during a negotiation. Agents are obliged to stick to this protocol, and deviations from the protocol are caught and penalized. This section discusses the details of the protocol for this assignment.

Agent A and B take turns in the negotiation. One of the two agents is picked at random to start. When it is the turn of agent X (X being A or B), that agent is informed about the action taken by the opponent. If the action was an Offer, agent X is subsequently asked to determine its next action and the turn taking goes to the next round.

If it is not an Offer, the negotiation has finished. The turn taking stops and the final score (utility of the last bid) is determined for each of the agents, as follows:

- the action of agent X is an Accept. This action is possible only if the opponent actually did a bid. The last bid of the opponent is taken, and the utility of that bid is determined in the utility spaces of agent A and B. The opponent is informed of this accept via the ReceiveMessage function (but now without the subsequent chooseAction).
- the action returned is an EndNegotiation. The score of **both** agents is set to 0.

So far for the protocol. If an agent does not follow this protocol, for instance by sending another action that is not one of the above or by crashing, that agent will get a utility of 0, and the opponent will be given the utility of his last bid (or 1 if he did not yet do a bid). If the agent deviates grossly from the protocol in such a way that it endangers the negotiator simulator itself, it may be disqualified entirely.

Time available for a Session

Normally, each negotiation session is allowed to last at most 120 seconds. If no agreement has been reached before this time, the negotiation will be terminated by killing the negotiation agents, and the utility of both parties will be 0. Only if (at least) one of the agents is a GUI agent requiring user input, the deadline is set to 1800 seconds.

Notice that manipulation of the available time (by delaying the response of your agent) can be an important factor influencing the negotiation results, and one

improvement for the example agent (SimpleAgent) would be to be more careful about this.

Negotiation Objects

The central data structures in the negotiation are the bid and the utilityspace. Both work in a domain.

Figure 1 shows an overview of the Domain and utility space data structures and their relations.

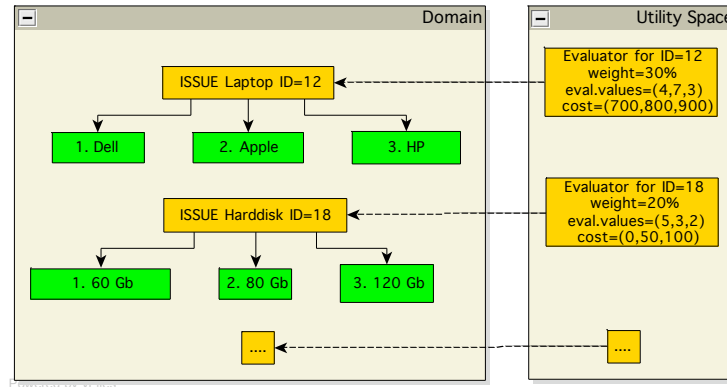


Figure 1. Overview of the datastructures and relations.

The **domain** describes which issues are the subject of the negotiation. To give a concrete example of a domain, in the laptop domain the domain is a list of issues, where the **issues** are laptop, hard disk and monitor. Issues are being referred to via the **issue ID**, a unique number for each issue. The domain description also describes the possible **values** that all the issues can take. In the laptop domain, all issues can have only discrete values, e.g. the hard disk issue can have only the values 60, 80 and 120. These issues are an instance of IssueDiscrete. There are other types of Issue but discussion of them falls out of the scope of this short discussion.

Issues are an instantiation of a more general **Objective** class. The objective class itself is not relevant except that some functions return an Objective, and the returned object then has to be cast to an Issue or IssueDiscrete as needed.

The **utilityspace** provides all the information enabling the computation of the utility of some bid. It is implemented as a list of evaluators, one evaluator for every issue in the domain. Because the issues in the laptop domain are all discrete issues, the evaluators are all EvaluatorDiscrete objects.

The **Evaluator** object contains a weight, and for each value that the issue can have it gives an evaluation-value and a cost value.

The **weight** indicates the relative importance of an issue. The sum of the weights of all issues is 1.0.

The **evaluation-value** gives the evaluation, or utility, associated with each value that an issue can take. For instance, for the hard disk issue above, the evaluation values could be 2 for the 60Gb, 4 for the 80Gb and 10 for the 120Gb, etc.

The exact formula for computation of the utilities is given in equation 1.

The **bid** is a set of values for each of the issues in the domain.

Utility of a Bid

A bid is a set of chosen values $v_1 \dots v_N$ for each of the N issues. Each of these values has been assigned an evaluation value $eval(v_i)$ in the utility space, and also there are fixed costs $cost(v_i)$ associated with each value.

The utility is the weighted sum of the normalized evaluation values, under the assumption that the cost is below the maximum cost of 1200. If the maximum cost is exceeded, the utility is zero.

$$Utility(v_1 \dots v_N) = \begin{cases} U(v_1 \dots v_N), & \text{if } CostSum(v_1 \dots v_N) < 1200 \\ 0, & \text{if } Cost(v_1 \dots v_N) > 1200 \end{cases}$$
$$U(v_1 \dots v_N) = \sum_{i=1}^N w_i \frac{eval(v_i)}{\max(eval(v_k))} \quad (1)$$
$$CostSum(v_1 \dots v_N) = \sum_{i=1}^N cost(v_i)$$

Thus, the utility function is more or less fixed, you can only adjust the weight values. This is done this way to allow a comparison of the results of the final negotiation results.

2. Installing the Environment

System Requirements

You can run the negotiation environment on any system running java version 5 or 6. You can download java from the internet (www.sun.com/java) [sun07].

Installation

To install the environment, the file `negotiator.zip` can be downloaded from Blackboard. Unzip the file at a convenient location on your machine.

This will result in a package containing the following files:

- **userguide.pdf**, containing this document
- **negosimulator.jar**, the negotiation simulator
- a templates folder, containing the **party** domain space (xml files).
- The **BayesianAgent.java** and **BayesianAgent.class** file

Progress & Error Messages

When you run the negoeditor (by double-clicking these applications), progress messages and error messages are printed mainly to the standard output. On Mac OSX you can view these messages by opening the console window (double-click on Systemdisk/Applications/Utilities/Console.app). On windows this is not directly possible. Console output can be read only if you start the application from the console window by hand, as follows. Go to the directory with the negosimulator and enter

```
java -jar negosimulator.jar
```

This will start the simulator, and all system.out messages will appear in the console window.

Bug reporting

The negotiation environment has been tested extensively on Mac OSX 10.9 and on Windows XP. It should run on any machine running java 1.5.0 or higher. This includes Solaris, Linux, and more recent versions of Mac OSX and Microsoft Windows.

There are still a number of known bugs in the negotiation environment, and possibly new bugs will be discovered during the course. If you find a bug, please report this to D.Tykhonov@tudelft.nl or W.Pasman@tudelft.nl.

4. Running Negotiations

To run a set of negotiation sessions, double click the `negosimulator.jar` file. The set-up screen pops up (Figure 2).



Figure 2. The negotiation set-up screen.

You need to set up the following items:

- The negotiation template. This is an xml file holding a description of the domain, plus the default values for all other fields in the set-up screen. You can edit the xml file by hand to change the default values. Alternatively, click the topmost "Browse" button and select a `domain.xml` file. For the running example with the laptop domain, select the `laptop_domain.xml` file.
- Number of sessions: the number of negotiation sessions that you want to run. Warning: do not type white spaces around the number.
- Agent A/B class name: the name of the negotiation agent that party A/B in the negotiation wants to use. You can use the provided agents `UIAgent` and `SimpleAgent`, or use your own agent placed in the directory containing the `negotiator.jar` file. `UIAgent` allows you to manually control party A/B in the negotiation. `SimpleAgent` is an agent that automatically handles the negotiation for party A/B.
- Agent A/B name: the plain name of the agent, used mainly when printing information about the progress and asking input.
- Agent A/B utility space: the xml file containing the utility space of agent A/B. Usually this is a file with a name like `laptop_A/B_utility.xml`.
- The Agent A starts Negotiation checkbox. Check this box if you want to start the negotiation with agent A (instead of at random).

The text field just above the start button will show real-time feedback on the ongoing negotiation. After these fields have been set appropriately, you can press the Start button to run the negotiation sessions.

After pressing start, a requester pops up (Figure 3) whether you want a utility plot. Press "Yes" to create such a plot, or no to proceed without.

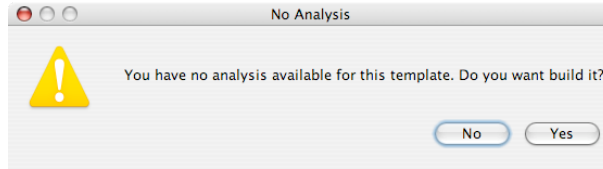


Figure 3. Request whether you want a utility plot.

Note, if you press "Show Model", you will end up in the utility space editor, editing the utility space of Agent A as discussed in Chapter **Error! Reference source not found.**

Using an Automatic Agent

If you selected an automatic negotiation agent, for instance BayesainAgent, there will not appear any window while that agent has the turn. The agent should pose its bid within a reasonable time. After the agent made its bid, the other agent is given the turn. If both agents are automatic, no windows will appear at all during the entire negotiation, and only the output windows will show the ongoing negotiation results.

Using the UIAgent

If you selected the agents.UIAgent for Agent A/B class name, the window as shown in Figure 4 will pop up every time input from negotiation agent A/B is needed.

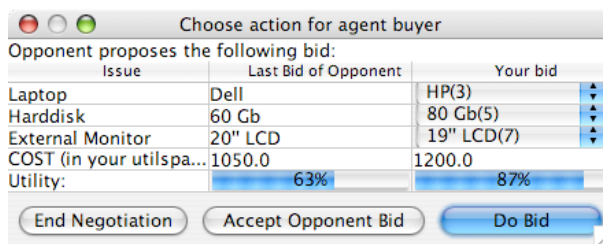


Figure 4. The GUI of UIAgent.

This GUI has three main components: the text field at the top level, the table showing the last bid and a possible next bid, and a row of buttons at the bottom.

The text field shows some text about the current negotiation state.

The table has three columns:

- The left column shows the names of the issues in the domain
- The center column shows the evaluation values for the issues as proposed in the last bid of the opponent (or "-" if this is the first round)
- The right column shows the current picked values for the issues. You can edit the current pick by clicking on the fields, which will open the combo boxes in the fields (Figure 5).

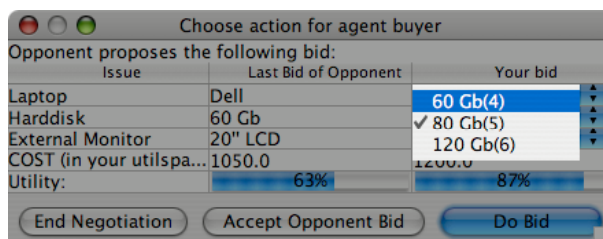


Figure 5. Combo box opens after clicking on a field, allowing change of the picked values for the next bid.

The last two rows of the table show the cost and utility of the last opponent's bid and your current bid. The cost field will turn red if you exceed the maximum cost of € 1200. The utility is shown as a percentage, and also as a bar of matching size. These values are computed according to your utility space, as that determines your score, and also because you have no access to the opponent's utility space. The lower three buttons allow you to submit the next bid as set in the right column, or to accept the opponent's last bid.

Negotiation Results

There are two types of results: the utility plot and the utility of the last bid for each round. The utility plot is shown only for the last round, while all results are collected in the score window.

The Utility Plot

If you enabled the analysis of the negotiation (Figure 3), a utility plot will be provided during after each negotiation completion. This plot looks as in Figure 6. The plot shows points corresponding to all possible bids, the Pareto Frontier (the greenish line), and the bidding progress plot.

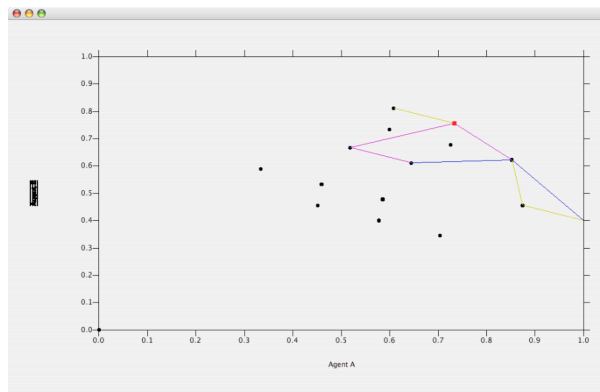


Figure 6. Utility plot. The plot also shows the Pareto Frontier (the greenish line), and the bidding sequence of agent A and agent B (the blue and the purple lines).

Score Window

The results of the negotiation are shown in the Negotiation Results screen (Figure 7). This screen shows four columns: the session number, the final utility for agent A and B, and the Protocol ErrorRemarks column. The utilities are the utilities of the final, accepted bid, as measured in the utility spaces of agent A and B (see the section Utility of a Bid). The Protocol ErrorRemarks button shows extra information in case one of the agents ignored the negotiation protocol, causing a non-standard scoring of the result (see the section Negotiation Protocol).

Negotiation results			
Session #	Utility for seller	Utility for buyer	Protocol ErrorRemarks
1	0.6077822994724837	0.8111693706922803	
2	0.6443245160860265	0.6110214965483346	
3	0.5995926003964769	0.7335191460912915	
4	0.6077822994724837	0.8111693706922803	
5	0.725620106089855	0.6779289437379261	
6	0.7338098051658619	0.7555791683389148	

Figure 7. Example of Negotiation score window. Here Agent A is the seller, agent B the buyer.

5. Writing a Negotiation Agent

This section discusses how you create an automatic negotiation agent. To explain this, the SimpleAgent.java code as provided in the installation package will be discussed.

It is assumed that you are familiar with programming in Java. In case you need more information about JAVA programming, please use the following link: <http://java.sun.com/docs/books/tutorial/index.html>. The Java API definitions can be found on <http://java.sun.com/j2se/1.5.0/docs/api/index.html>.

In the first place, a negotiation agent has to extend the negotiator.agents.Agent class. Table 1 shows the fields and methods of this class.

Table 1. Methods and fields of the Agent class

String fName	the name of the agent
UtilitySpace utilitySpace	the current utility space. This usually is an instance of the UtilitySpace that you specified in the negotiation set-up screen (Figure 2)
Date startTime	the start time for the current negotiation session
Integer totalTime	the total available time (seconds) to complete the current negotiation session
void init (int sessionNumber, int sessionTotalNumber, Date startTimeP, Integer totalTimeP, UtilitySpace us)	
void ReceiveMessage (Action opponentAction)	
Informs the agent which action the opponent did	
Action chooseAction ()	
This function should return the action your agent wants to make next. This function is called immediately after a ReceiveMessage, and only if the opponent made an Offer or if this is the first round in the session.	
String getName ()	returns the name of the agent

By extending Agent, your agent can access the fields as it likes.

To implement your agent, you have to override two or three three classes:

- **public void** ReceiveMessage(Action opponentAction)
- **public** Action chooseAction()
- **public void** init (int sessionNumber, int sessionTotalNumber, Date startTimeP, Integer totalTimeP, UtilitySpace utilspace)

Init

An important consideration for the implementation is that an agent may participate in multiple negotiation sessions with the same opponent. This enables the agent to learn from the previous sessions. For this reason, the negotiation environment calls the method init before starting the new session.

The parameter `sessionNumber` is a unique number identifying the current negotiation session.

The `sessionTotalNumber` informs you about the total number of sessions.

The `startTimeP` is the time at which this session was started (which can be before the call to the `init` function).

The `totalTimeP` is the total number of seconds available to complete this negotiation session.

Finally, `utilspace` provides the agent with his utility space that he has to work with.

Overriding the `init` is optional. If it is not overridden, the default `init` will save the `fName`, `startTime`, `totalTime` and `utilitySpace`. If you also want to save the `sessionNumber` and `sessionTotalNumber` you have to override the `init` function, similar to the `SimpleAgent.java` example:

```
private int sessionNumber;
private int sessionTotalNumber;

public void init(int sessionNumberP, int sessionTotalNumberP,
    Date startTimeP, Integer totalTimeP, UtilitySpace us)
{
    super.init (sessionNumberP, sessionTotalNumberP, startTimeP,
        totalTimeP, us);
    sessionNumber = sessionNumberP;
    sessionTotalNumber = sessionTotalNumberP;
}
```

ReceiveMessage

The `ReceiveMessage(Action opponentAction)` informs you that the opponent just did `opponentAction`. The `opponentAction` may be null if you are the first to place a bid, or an `Offer` containing the bid of the opponent. It may also be an `Accept` or `EndNegotiation` action.

The `chooseAction()` asks you to return an `Action` to make the next step in the negotiation.

In the `SimpleAgent` code, the following code is available for `ReceiveMessage` (Figure 8). This will be the typical code for automatic negotiation agents.

```
public void ReceiveMessage(Action opponentAction)
{ actionOfPartner = opponentAction; }
```

Figure 8. Example code for ReceiveMessage

ChooseAction

Figure 9 shows the example code for the `chooseAction` method. For safety, all code was wrapped in a try-catch block, because if our code would accidentally contain a bug we still want to return a good action (failure to do so is a protocol error – see Negotiation Protocol – and results in a score of 0 for us!).

The sample code works as follows. If we are the first to place a bid, we place a random bid with sufficient utility (see the `.java` file for the details on that). Else, we

determine the probability to accept the bid, depending on the utility of the offered bid and the remaining time. Finally, we randomly accept or pose a new random bid.

```

public Action chooseAction()
{
    Action action = null;
    try {
        if(actionOfPartner==null)
            action = chooseRandomBidAction();
        if(actionOfPartner instanceof Offer)
        {
            Bid partnerBid = ((Offer)actionOfPartner).getBid();
            double offeredutil=
                utilitySpace.getUtility(partnerBid);
            double time=((new Date()).getTime()-
                startTime.getTime())/(1000.*totalTime);
            double P=Paccept(offeredutil,time);
            if (P>Math.random()) action = new Accept(this);
            else action = chooseRandomBidAction();
        }
        Thread.sleep(1000); // just for fun
    } catch (Exception e) {
        System.out.println("Exception in
        ChooseAction:"+e.getMessage());
    }
    action=new Accept(this);
    return action;
}

```

Figure 9. Example code for chooseAction

The Paccept function is a probabilistic acceptance function where P equals

$$P_{accept} = \frac{u - 2ut + 2\left(t - 1 + \sqrt{(t-1)^2 + u(2t-1)}\right)}{2t-1} \quad (2)$$

Where u is the utility of the bid made by the opponent (as measured in our utility space), and t is the current time as a fraction of the total available time.

Figure 10 shows how this function behaves depending on the utility and remaining time. It can take quite some time to figure out a formula that suits the requirements, but it is at the heart of the example agent.

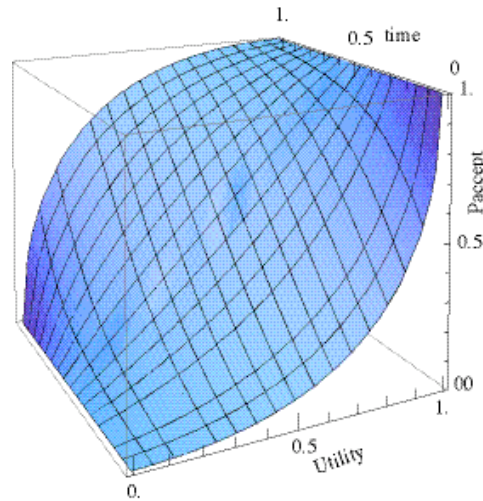


Figure 10. Paccept value as function of the utility and time (as a fraction of the total available time)

The UtilitySpace and its functions should be considered as 'given'. You may want to override the init function, to catch the sessionNumber and sessionTotalNumber. See the SimpleAgent.java example file. Automatic agents have to negotiate on their own, and are not allowed to communicate with a human user. Therefore, do not override the isUIAgent function in automatic negotiation agents.

Compiling an Agent

To compile the agent, you put YourAgent.java code in the directory containing the negotiator.jar file, and use the command line

```
javac -cp negotiator.jar YourAgent.java
```

After compilation, the resulting YourAgent.class file can be loaded into the negotiator simulator by typing "YourAgent" (fill in your actual agent name) in the negotiation set-up screen (Figure 2).

6. Data structures

This section discusses the data structures that are relevant when writing a negotiation agent. Only the functions that fetch information out of the available data structures will be discussed, as modification of these data structures is strongly discouraged in this context.

The following sections discuss the methods of the relevant objects in more detail. For a number of data structures the constructor is not given, as the readily provided objects should be used.

Domain

Objective <code>getObjective(int ID)</code> ID is the ID of the objective returns the objective with the given ID.
Bid <code>getRandomBid()</code> returns a bid with randomly set values. The bid may have utility 0.
ArrayList<Issue> <code>getIssues()</code> get all issues as an arraylist.

ValueDiscrete (implements Value)

ISSUETYPE <code>getType()</code> returns ISSUETYPE.DISCRETE.
String <code>getValue()</code> returns the String associated with this Value

IssueDiscrete (implements Issue)

int <code>getNumber()</code> returns the issueID of this issue.
String <code>getName()</code> returns the name of the issue
int <code>getNumberOfValues()</code> returns the number of values available for this issue.
String <code>getDescription()</code> returns the description for this issue.
ValueDiscrete <code>getValue(int n)</code> returns the n^{th} value of this issue. n can be $0..getNumberOfValues()-1$.
String <code>getStringValue(int n)</code> returns the n^{th} value of this issue as a String.
int <code>getValueIndex(String s)</code> returns the index number n for which <code>getStringValue(n)=s</code> .
ArrayList<ValueDiscrete> <code>getValues()</code> returns an ordered list with all values associated with this issue.

UtilitySpace

Evaluator **getEvaluator**(int index)

index is the IDnumber of the issue.

returns an evaluator for the objective or issue.

double **getUtility**(Bid bid) **throws** Exception

computes the utility of a given bid in your utility space. This is in fact the weighted sum of **getEvaluation**().

Can throw an exception if there is a problem with the computations, for instance if the utility space contains illegal weights or evaluators.

double **getEvaluation**(int pIssueIndex, Bid bid)

throws Exception

Computes the utility of one of the issues (pIssueIndex) in a bid. This just translates to a call to **EvaluatorDiscrete.getEvaluation**.

boolean **constraintsViolated**(Bid bid)

Check that the constraints are not violated in this bid. Returns true if the bid violates the constraints. In the current negotiator system, the only, hard coded constraint is that $\text{getCost}(\text{bid}) \leq 1200$. This constraint is used with every utility space.

Bid **getMaxUtilityBid**() **throws** Exception

returns a bid with the maximum utility value attainable in this utility space.

Throws an exception if there is no bid at all in this utility space

double **getWeight**(int issueID)

returns the weight of the issue with given issueID

Objective **getIssue**(int index)

does exactly the same as **Domain.getIssue**

Domain **getDomain**()

returns the domain

Double **getCost**(Bid bid) **throws** Exception

returns the cost of the given bid. throws if cost can not be computed for some reason.

Bid

fDomain

variable holding the domain

Bid(Domain domainP, HashMap<Integer,Value> bidP)

throws Exception

constructor for a Bid.

The bidP is a hashmap that maps issue IDs to Values (which are Strings).

Will throw exception if not all issues in the domain are assigned a value.

Value **getValue**(int issueNr) **throws** Exception

returns the picked value for a given issue ID number.

void **setValue**(int issueId, Value pValue)

sets the value for issue with issueId to pValue.

boolean **equals**(Bid pBid)

returns true if bids are equal

HashMap<Integer, Value> **getValues**()

returns the entire set of bid values

EvaluatorDiscrete (implements Evaluator)

double getWeight()
returns the weight for this evaluator, a value in [0,1].
Integer getValue (ValueDiscrete alternativeP)
returns the non-normalized evaluation-value may return null if the alternativeP is not a value, or when the value has not been set.
Integer getEvalMax() throws Exception
returns the largest evaluation value available. Throws if there are no evaluation values available.
Double getEvaluation (UtilitySpace uspace, Bid ID, int index) throws Exception
returns the normalized evaluation value ($eval(bid \text{ for issue } ID)/\max(eval(v_k))$ in equation 1)
Double getEvaluation (ValueDiscrete v) throws Exception
returns the normalized evaluation value of value v ($eval(v)/\max(eval(v_k))$ in equation 1)
Double normalize (Integer EvalValue) throws Exception
EvalValue is the evaluation value to be normalized returns $EvalValue/\max(eval(v_k))$ throws if the EvalValue is null, or if the normalization fails.
Double getCost (Value value)
value is the value of which the cost is needed returns the cost for given value, or null if no cost available for value.
double getMaxCost()
returns the maximum of the costs for this evaluator.
Double getCost (UtilitySpace uspace, Bid bid, int id) throws Exception
Returns the cost of issue with given id. May throw if the bid is incomplete or utilityspace has problems
Value getMaxValue()
returns the first value that has the maximum evaluation-value.
Value getMinValue()
returns the value that has the smallest evaluation-value

References

[sun07] Sun Developer Network, JDK 6 Update 3.
<http://java.sun.com/javase/downloads/index.jsp>

